Visual Performance Analysis of Memory Behavior in a Task-Based Runtime on Hybrid Platforms

Lucas Leandro Nesi, Samuel Thibault, Luka Stanisic, Lucas Mello Schnorr

Visual Performance Analysis of Memory Behavior in a Task-Based Runtime on Hybrid Platforms

Lucas Leandro Nesi*, Samuel Thibault†, Luka Stanisic‡, Lucas Mello Schnorr*
* Institute of Informatics/PPGC/UFRGS, Porto Alegre, Brazil
† Inria Bordeaux Sud-Ouest, Bordeaux, France
‡ Max Planck Computing and Data Facility, Garching, Germany

***Abstract*—Programming parallel applications for heterogeneous HPC platforms is much more straightforward when using the task-based programming paradigm. The simplicity exists because a runtime takes care of many activities usually carried out by the application developer, such as task mapping, load balancing, and memory management operations. In this paper, we present a visualization-based performance analysis methodology to investigate the CPU-GPU-Disk memory management of the StarPU runtime, a popular task-based middleware for HPC applications. We detail the design of novel graphical strategies that were fundamental to recognize performance problems in four study cases. We first identify poor management of data handles when GPU memory is saturated, leading to low application performance. Our experiments using the dense tiled-based Cholesky factorization show that our fix leads to performance gains of 66% and better scalability for larger input sizes. In the other three cases, we study scenarios where the main memory is insufficient to store all the application's data, forcing the runtime to store data out-of-core. Using our methodology, we pin-point different behavior among schedulers and how we have identified a crucial problem in the application code regarding initial block placement, which leads to poor performance.**

I. Introduction

A challenge found in the High-Performance Computing (HPC) domain is the complexity of programming applications. The task-based programming paradigm presents numerous benefits, and many researchers believe this is currently the optimal approach to program for modern machines. The tasking related extensions to the OpenMP (in 4.0 and 4.5 version), and the upcoming 5.0 standard with even more features confirm this trend. In general, a task-based approach is extremely efficient for load-balancing and intelligently using all the resources' computational power in heterogeneous platforms. It transfers to a runtime some activities that are usually carried out by programmers, such as mapping compute kernels (tasks) to resources, data management, and communication. Task-based applications use a Direct Acyclic Graph (DAG) of tasks as the main application structure to schedule them on resources, considering tasks dependencies and data transfers. Among many alternatives like Cilk [1], Xkaapi [2], and OmpSs [3]; StarPU [4] is one example of a runtime using this paradigm. Its features include the use of distinct tasks' implementations (CPU, GPU), different tasks schedulers, and automatically managing data transfers between resources.

The performance analysis of task-based parallel applications is complicated due to its inherently stochastic nature regarding variable task duration and their dynamic scheduling. Different performance analysis methods and tools can be used to aid on this matter, including analytical modeling of the task-based application theoretical bounds [5] and the application-runtime simulation which allows reproducible performance studies in a fully-controlled environment [6], [7]. StarPU can also collect execution traces that describe the behavior of the application enabling other tools to provide information for the performance analysis. Possible uses of the information provided by the runtime can be in the form of performance metrics (number of ready and submitted tasks, the GFlops rate, etc.), the indication of poor behavior (i.e., absence of work in the DAG critical path), or visualization techniques (panels that illustrate the application and the runtime behavior over time). The visualization-based approach can combine all these investigation methods to facilitate the analysis with graphical elements. The StarVZ workflow [8] is an example of a visualization tool that leverages application/runtime traces. It employs consolidated data science tools, most notably `R` scripts, to create meaningful views that enable the identification of performance problems and testing of what-if scenarios.

Interleaving data transfers with computational tasks (data prefetching) is another technique that has a significant impact on performance [9]. The goal is to efficiently manage data transfers among different memory nodes of a platform: main (RAM), accelerator (GPUs), and out-of-core (hard drive) memories. Factors like the reduction of data transfers between heterogeneous devices and host, better use of cache, and smarter block allocation strategies play an essential role for performance. Simultaneously, many applications require an amount of memory greater than the available RAM. These applications require the use of out-of-core methods, generally because disk memory is much larger than main memory [10]. Correctly handling which data blocks stay in main or disk memory is a challenge. The complexity of evaluating these memory-aware methods motivates the design of visualization-based performance analysis techniques tailored explicitly for data transfers and general memory optimizations.

In this paper, we focus on the analysis of the StarPU's memory management performance using trace visualization. They enable a general correlation among all factors that can impact the overall performance: the application algorithm, the runtime decisions, and memory utilization. The main contributions are the following. **(a)** We extend the StarVZ workflow

by adding new memory-aware visual elements that help to detect performance issues in the StarPU runtime and the task-based application code. **(b)** StarPU is augmented with extra trace information about the memory management operations, such as new memory requests, additional attributes on memory blocks and actions, and data coherency states. **(c)** We present the effectiveness of our methodology with four scenarios that use the dense linear algebra solver Chameleon [11]. In the first case, we show how we identified a problem inside the StarPU software, and compare the application performance after our proposed correction patch. In the second case, we analyze the idle times when using out-of-core. In the third case, we offer an alternative method on the application to allocate blocks in out-of-core memory in a more efficient way. In the last case, we study the memory/application behavior between the `DMDAS` and `DMDAR` schedulers. These methods lead to a reduction of $\approx$66% in the execution time when using a heterogeneous platform composed of CPUs and GPUs. Although we use the methods on StarPU, they are general and extendable to other runtimes.

The paper is structured as follows. Section II provides basic concepts on the StarPU runtime system and the dense linear algebra Cholesky factorization as implemented by Chameleon. Section III presents related work on the visualization of memory management and task-based applications. We also discuss our approach against the state-of-the-art. Section IV presents the visual-based methodology to investigate the performance of memory operations in the StarPU runtime, employing a modern data science framework. Section V details the experiments conducted in four test cases. Section VI discusses the limitations of our strategy and Section VII concludes this paper with future work. The companion material of this work is publicly available at https://doi.org/10.5281/zenodo.2605464.

II. Background Concepts

We provide a general overview of the StarPU runtime and a detailed explanation of how the Chameleon project implements a dense tiled-based Cholesky factorization using a Directed Acyclic Graph (DAG) of tasks for heterogeneous platforms.

A. The StarPU runtime

The StarPU runtime uses the Sequential Task Flow (STF) model [12], where tasks are sequentially submitted during the application execution and are dynamically scheduled to workers. In such a model, there is no need to unroll the whole Directed Acyclic Graph (DAG) of tasks before starting tasks execution. StarPU tasks might have multiple implementations, one for each type of resource (such as x86 CPUs, CUDA GPUs, and OpenCL devices), and must register memory handles to identify the memory blocks on which they read and write data. Depending on resource availability and the heuristic, the scheduler dynamically chooses one of the task versions and puts it to execute. StarPU employs different heuristics to allocate tasks to resources. Classical heuristics are `LWS` (local work stealing) and `EAGER` (centralized deque). More sophisticated schedulers consider additional information. The `DMDA` (deque model data aware) scheduler, for example, uses estimated task completion time and data transfer time to take its decisions [9]. Another example is the `DMDAR` (deque model data-aware ready) scheduler; that additionally considers memory handles already present on the workers.

The runtime is also responsible for transferring data between resources, for controlling the presence and the coherence of the memory handles. StarPU creates one memory manager for each different type of memory. For example, there is one memory manager for the `RAM` associated with one NUMA node (shared by all CPU cores on that socket), one for each GPU, and so on. StarPU adopts the basic MSI protocol, with the states `Modified/Owned`, `Shared`, and `Invalid`, to manage the state of each memory handle on the different memories. At a given moment, each memory block can assume one of the three states on the memory managers [4]. When a task is scheduled, StarPU will internally create a memory request for one of the tasks memory dependencies to the chosen resource. These requests are handled by the memory managers that are responsible for allocating the block of data and issuing the data transfer. When tasks are scheduled well in advance, StarPU prefetches data, so the transfers get overlapped with computations of the ongoing tasks [13].

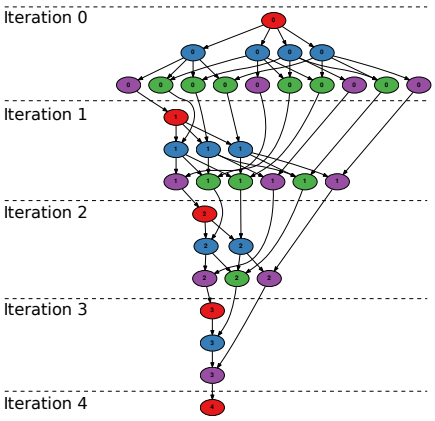
Furthermore, recent versions of StarPU support the use of out-of-core memory (disk i.e., HDD, SSD) when `RAM` occupation becomes too high. The runtime employs a Least-Recently-Used (LRU) algorithm to determine which data blocks should be transferred to disk to make room for new allocations on `RAM`. Interleaving such data transfers with computation and respecting data dependencies on the critical path is fundamental to good performance.

B. The Chameleon Package

The Chameleon package [11] contains a series of dense linear algebra solvers implemented using the sequential task-based paradigm. From the set of available solvers, we adopt the task-based solver that implements the dense linear algebra Cholesky factorization on top of the StarPU runtime, because many HPC applications used it as a computing phase. The Cholesky factorization algorithm runs over a triangular matrix divided into blocks, using four different tasks: `dpotrf` (Cholesky Factorization), `dtrsm` (Triangular Matrix Equation Solver), `dsyrk` (Symmetric Rank-k Update) and `dgemm` (Matrix Multiplication), as shown in Figure 1a. The task-based Cholesky factorization divides the input matrix into tiles (blocks), making each task associated with a block. The factorization essentially begins with tasks on lower coordinates blocks and iteratively computes all matrix blocks for all coordinates. The Figure 1b demonstrates the resulting DAG for a matrix divided into 25 blocks (N = 5). The Chameleon framework generates the full matrix to conduct numerical checks. Since in our case the solver is used independently of real application code, the Chameleon testing code includes an input generation task called `plgsy` to create double floating-point values for the matrix tiles.

```
for (k = 0; k < N; k++) {
   DPOTRF(RW, A[k][k]);
   for (i = k+1; i < N; i++)
     DTRSM(RW, A[i][k], R, A[k][k]);
   for (i = k+1; i < N; i++) {
     DSYRK(RW, A[i][i], R, A[i][k]);
     for (j = k+1; j < i; j++)
        DGEMM(RW, A[i][j], R, A[i][k],
                           R, A[j][k]);
   }
}
```

(a) The Cholesky Algorithm.



(b) Corresponding DAG for $N = 5$.

Fig. 1: The tiled Cholesky code and the DAG for $N = 5$.

III. Related Work

There are very few performance analysis tools focused on the performance analysis of DAG-based HPC applications, such as Haugen *et al.* [14], DAGViz [15], grain graphs [16] (for OpenMP applications), and Temanejo [17]. Most of them provide either the view of the DAG without temporal information or space/time view showing how tasks are scheduled to workers. More traditional tools [18], [19], [20], [21] focused on MPI+OpenMP or MPI+CUDA programming models can also be employed, but they lack the fundamental ability to consider critical path analysis or task dependencies delays as indicated by the DAG. More recently, StarVZ [8] has been proposed as an extensible R-based framework for the performance analysis of task-based HPC applications. It includes several visualization panels enriched with theoretical bounds and task-dependencies delays that correlate observed performance with the DAG. Even if some of these tools provide unwavering DAG-oriented support, they generally lack a specific methodology to analyze the impact of different block allocation policies on application performance.

More recently, Ceballos *et al.* [22] propose TaskInsight, a tool to evaluate how data reuse among application tasks might affect application performance. They quantify the variability of task execution by measuring data-aware hardware counters (i.e., cache misses) of some tasks when another task scheduling is being carried out. Despite their focus on such kind of memory interference, they overlook the impact of the application DAG and the effects of data prefetching and possible data transfers between different types of devices that are fundamental in current multi-GPU platforms. Miquel *et al.* [23] also study data transfer operations focusing on data reuse in task-based runtimes. The authors propose the Kernel Reuse Distance (KRD) metric which measures the amount of data reuse on caches with different sizes. They consider the reuse of multiple cores that access the same levels of caches. The KRD metric is derived from data memory access traces and can be used to understand the quality of data reuses on the applications. Although this metric can be used to explain performance differences in some situations, more events could be collected from traces to provide a better view of the application memory.

Our approach provides a multi-level performance analysis of data management operations on a heterogeneous multi-GPU and multi-core platform. We combine a high-level view of the application DAG with the low-level runtime decisions, which guides us in identifying and fixing performance problems. Instead of only using low-level metrics and comparing them with multiple executions, we focus on the behavior understanding of representative executions. We also design visualization elements specifically for the performance analysis of memory transfers in a DAG-based runtime, enriching our perception of task-based applications running on heterogeneous platforms.

IV. Memory-Aware Visualization Panels

We present our methodology to investigate the memory manager behavior and memory block allocations on resources. StarPU's data management module is responsible for all actions involving the application's memory. While absent from the original StarPU code, we have added events to the runtime's tracing mechanism to track the data management system. As a consequence, we proposed a set of extensions to gather the necessary information needed for our performance analysis. We first include the events' memory identification on all events with extra information to allow correlations between runtime activities and to understand the decisions behind it. Second, we trace the memory's coherence update function to keep track of the whereabouts of a memory block along the execution. Third, we track all memory requests (prefetch, fetch, allocation, sync) carried out by the runtime. The capture of traces is a feature already present in the StarPU runtime, and we extend it to add new information.

Our memory-aware visualization panels are designed to leverage these extra behavioral data about memory activities. The presence of memory blocks on each memory manager is used to understand the general behavior of the application. In what follows, we detail our data-aware visualization strategies.

A. Enriched Memory-Aware Space/Time View

Employing Gantt-charts to analyze parallel application behavior is very common. It is used to show the behavior of observed entities (workers, threads, nodes) along time. We have adapted and enriched such kind of view to inspect

the memory manager behavior, as shown in the example of Figure 2. On the Y-axis, the figure lists the different memory managers associated to different device memories: `RAM`, different accelerators (memory of GPU and OpenCL devices), and permanent storage in the case of out-of-core (`OOC`) disk memory. In this example, we have only three memory managers: `RAM`, `GPU1`, and `GPU2`. The plot presents the actions of each manager over time with colored rectangles tagged with block coordinates (i.e., for `GPU2`: 1×3, 0×2, and so on) from the application problem. The rectangles in this figure mainly represent different `Allocating` states carried out by those managers, except for the `RAM` manager that had no registered behavior in the depicted 10ms interval. In the right of each manager line, the panel describes the percentage of time of the most recurring state using the same color. For instance, the `GPU2` manager spent 75.15% of the time of this specific time interval in the `Allocating` state.

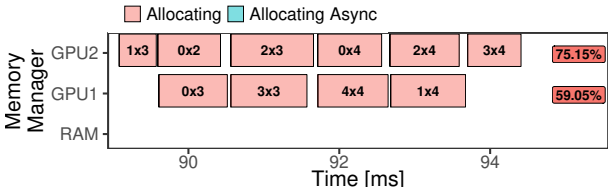


Fig. 2: The behavior of three memory managers (`RAM`, `GPU1`, `GPU2`) showing allocations states for different memory blocks.

B. Block Residency in Memory Nodes

A given block coordinate of an HPC application (i.e., Cholesky factorization) may reside in multiple memory nodes along the execution. For example, there can be many copies of a given block if workers executing on different devices perform read operations only. This observation is due to the adoption of the MSI protocol by StarPU, where multiple memory nodes have copies of the same memory block (see Section II for details). Figure 3 represents the location of a given memory block along the execution. Each of the five facets of the Figure represents one memory block with the coordinates 0×0, 0×1, 0×2, 0×3 and 0×4 of the input matrix. For each block, the X-axis is the execution time, discretized in time intervals of 20ms. This interval is sufficiently large for the visualization and small enough to show the application behavior evolution. At each time interval, the Y-axis shows the percentage of time that this block is on each memory node (color). For example, if a block is first owned by `RAM` for 18ms and then for 2ms by `GPU2`, the bar will be 90% blue and 10% yellow. Since each block can be shared and hence present on multiple memory nodes, the maximum residency percentage on the Y-axis may exceed 100%. Moreover, if the memory resides for only a portion of the interval, the percentage would be less than 100%.

With this new visualization, we can check a summarized evolution of data movement and resource's memory utilization. For example, Figure 3 details that the memory block with coordinates 0×0 stayed in `RAM` throughout the execution, while other blocks remain in `RAM` only until ≈80ms of the execution. We are capable to quickly spot anomalies by correlating the block coordinates residence with the application phases. Very frequently in linear algebra, a lower block coordinate is only used at the beginning of the execution, so it should be absent after the initialization phase (which would be demonstrated as 0% occupancy of that block after it is no longer needed).

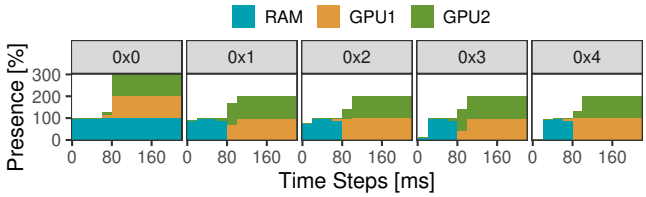


Fig. 3: The location of memory blocks along time.

C. Detailed Temporal behavior of Memory Blocks

The previous panel (see Figure 3) shows where a given block is located (on which memory node) throughout the execution. Figure 4, besides showing the memory block location, additionally depicts all the runtime and application tasks activities that affect the block behavior. Here, we employ the traditional Gantt-chart as a basis for the visualization, where the X-axis is the time in milliseconds, and the Y-axis represents the different memory managers. There are two types of states, depicted as colored rectangles. The ones shown in the background with a more considerable height represent the residency of the memory block on the managers: the red color expresses when a memory node is an owner, while the blue color indicates the block is shared among different managers. The inner rectangles represent the Cholesky tasks (`dpotrf`, `dtrsm`, `dsyrk`, `dgemm`, and `dplgsy`) that are executing and using that memory block from that memory manager. We augmented the representation with different events associated with the memory blocks on the respective manager and time. The circles (`Allocation Request`, `Transfer Request`) are either filled or unfilled, for fetch or prefetch operations, respectively. The arrows are used to represent a data transfer between two memory nodes and have a different meaning (encoded with different colors: intra-node prefetch and fetch). Finally, two vertical lines indicate the correlation (last dependency and last job on the same worker) with a specific application task that one in this example wants to study. Here, we highlight the task ID 90 (which is a `dsyrk` task). The green vertical line represents the end of the last dependency that releases task 90, and the yellow represents the end of the last task executed on the same worker.

D. Block Coordinates Animation to track Allocation History

The application running on top of StarPU determines the data and the tasks that will be used by the runtime. Instead of only considering the utilization of resources, we want to correlate the algorithm and the runtime decisions. We are then interested in a view that takes into account the coordinates

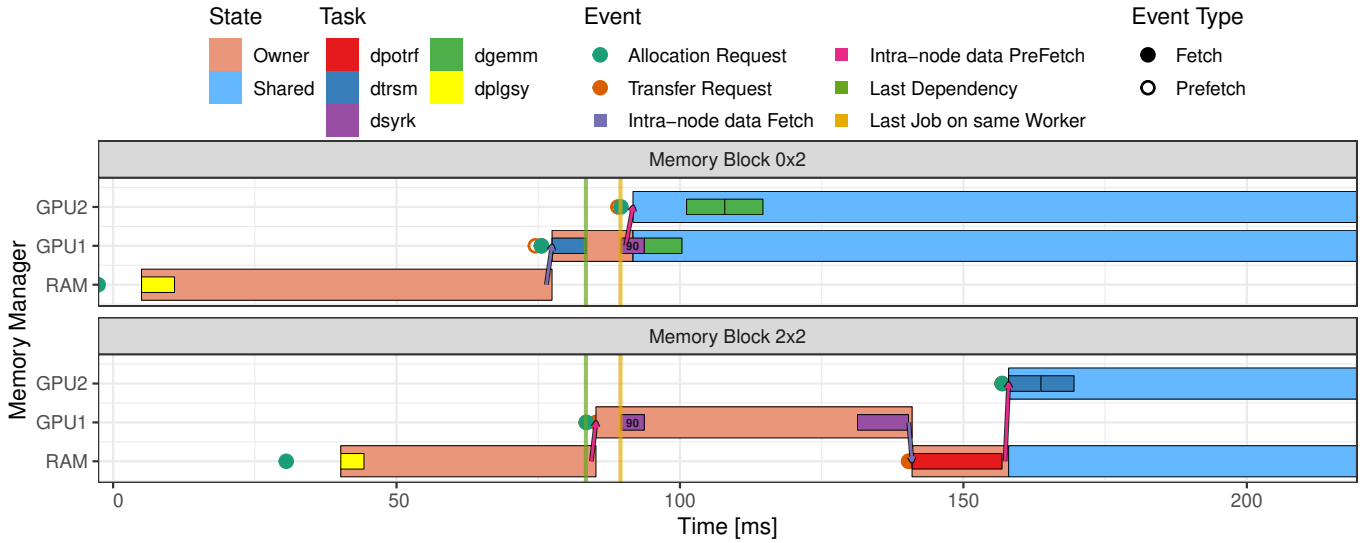


Fig. 4: The detailed view of all events related to a memory block (facet), showing residence (larger height rectangles) and the use by application tasks (inner rectangles). The figure shows the case for two memory blocks: 0×2 (top) and 2×2 (bottom).

of the blocks in the original data, illustrating which task is using each block, and their state on the managers (owned, private or shared). Figure 5 depicts a snapshot of all memory blocks locations and the running tasks in a specific time. The visualization has three facets, one for each of memory managers (`RAM`, `GPU1`, `GPU2`). Each manager has a matrix with the block coordinates in the X and Y-axis. On this matrix, each colored square represents one memory block. The colored inner squares (write mode access) or circles (read) inside those blocks represent application tasks. With this visualization, it is easy to confirm how the memory data flow correlates with the blocks position. In Figure 5, for example, we can see that only two blocks are on `RAM` and that both GPUs share the first row. Moreover, there is a `dpotrf` task executing over block 1×1 in `RAM` and a `dgemm` task on each GPU. `GPU1` has write access on the `dgemm` task on block 1×2, and two read accesses on blocks 0×1 and 0×2. By stacking consecutive snapshots, we can create an animation that shows the residence of memory blocks along time. This feature is particularly useful to understand the algorithm behavior and the data allocation policy. As of now, StarPU developers are integrating such a view for general use.

E. Heatmap to verify Memory Block residence along time

Apart from the previous memory snapshot visualization, we are also interested in an execution overview of the handles locality among the managers. Our final panel consists of a traditional heat map visualization to provide a summary of the total presence of the tiles on each manager. Figure 6 depicts an example. There is one visualization facet for each memory manager, and each square represents a memory block positioned on its application matrix coordinate. The blue color tonality represents the total amount of time that the block is present on the manager. In the Figure 6, for example, we can see that all blocks of the diagonal stood more time on `RAM`

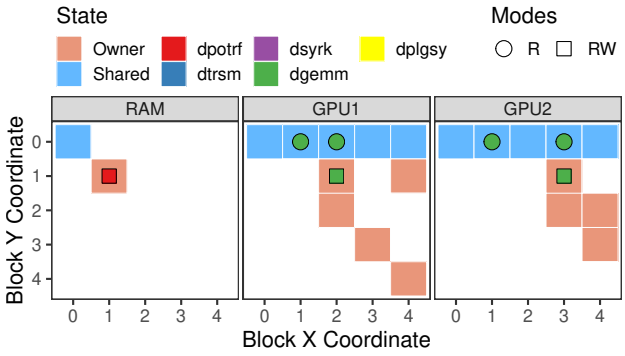


Fig. 5: A snapshot of memory block residence.

compared to other blocks (because for Cholesky, `dpotrf` and `dsyrk` tasks on the diagonal are typically executed on CPUs, to let GPUs process mostly SYRK and GEMM tasks).

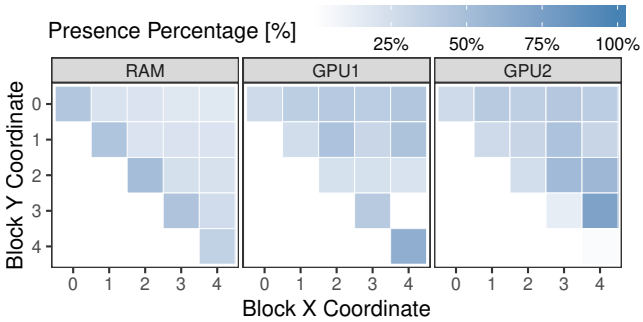


Fig. 6: Per-manager heatmaps for memory blocks presence.

V. Experimental Results with Dense Cholesky

We use a host equipped with an Intel Xeon CPU E5-2620 with eight physical cores, 64 GB of DDR4 RAM, two NVIDIA

GeForce GTX 1080Ti 11GB (driver version 396.24, CUDA version 9.2.88), and a 2TB SSD as the auxiliary memory for the out-of-core experiments. The StarPU and the Cholesky code from the Chameleon project have been compiled with GCC 7.3.0 using the default configurations. The Cholesky runs use a block size of 960x960. We use a Debian 4.16.12 for the first scenario and Ubuntu 18.04 for the other three cases.

We describe **(A)** how we have identified a runtime's wrong perception of the total used memory in GPU, causing StarPU to issue numerous allocation requests that impact on performance. We investigate **(B)** why the workers were presenting idle times when using the out-of-core feature and possible internal insights on how to solve it. In the last two scenarios, we compare **(C)** how the `LWS` and the `DMDAR` schedulers get affected by the matrix input generation with limited `RAM` memory and OOC, and **(D)** how `DMDAR` and `DMDAS` behave in a CPU-only setup with very limited `RAM`.

A. *Erroneous control of total used memory of GPUs*

Preliminary tests indicated that the dense Cholesky factorization of the Chameleon solver had performance problems when using an input matrix larger than the GPU memory size. When applying the overview methods described in Section IV, we were able to identify that GPUs are generally idle. This problem motivated us to investigate this issue further using other views of our methodology. In what follows, we describe the identification of the problem and the resolution.

Figure 7 presents, from top to bottom, the StarVZ plots: **(a)** Application Workers, **(b)** Memory Manager, **(c)** StarPU Workers, **(d)** Ready Tasks, and **(e)** Used memory. In **(a)** and **(c)** the Y-axes represent the workers (CPU cores and GPU devices), in **(b)** the memory nodes (in this case, `GPU1`, `GPU2`), in **(d)** the number of ready tasks, and in **(e)** the per-GPU memory utilization in MB. In all plots, the X-axis is the time in milliseconds (ms). Each state has a different color associated with its task or action. The red vertical dashed line has been later manually added to emphasize the moment where the total GPU used memory reaches a plateau with its maximum value. At that moment, we can perceive that the GPUs spend a lot of time idle (`GPU1` with 33%, `GPU2` with 32%), which is impairing the overall application performance.

The possible correlation between idle times and allocation states led us to investigate actions of the memory manager after the memory utilization reached its maximum. We select an arbitrary time frame since the behavior is similar after achieving the memory utilization peak. Figure 8 provides a temporal zoom (≈50ms period) on the Memory Manager panel, depicting multiple `Allocating` actions (red rectangles) for different memory block coordinates of the input matrix. We can see that there are many allocating attempts of the same memory blocks occurring many times. This repeated behavior is considered a problem because we would expect a single allocation request is enough. This analysis leads us to investigate the StarPU source code responsible for allocation, which indicated that the runtime keeps trying to allocate blocks indefinitely until it gets a successful allocation. Using the GPU

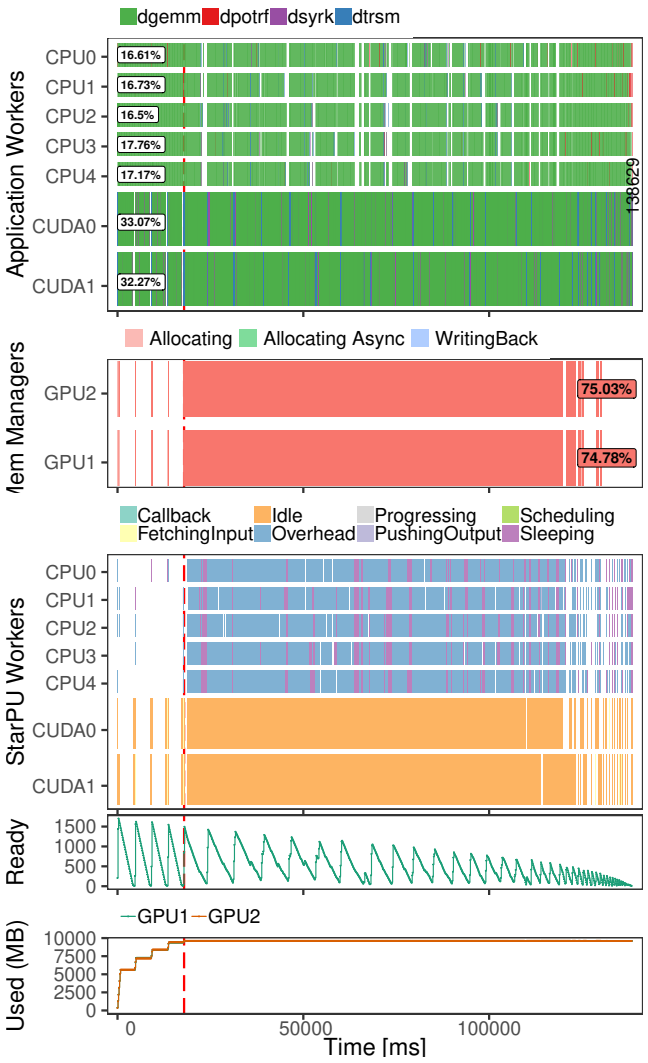


Fig. 7: Multiple Performance Analysis plots.

resources monitor, we confirmed that the GPUs are using all the memory. Our hypothesis at this point is that the devices do not have enough memory, but this should not be a problem, as StarPU could free multiple memory blocks, especially those that would no longer be used by the Cholesky factorization.

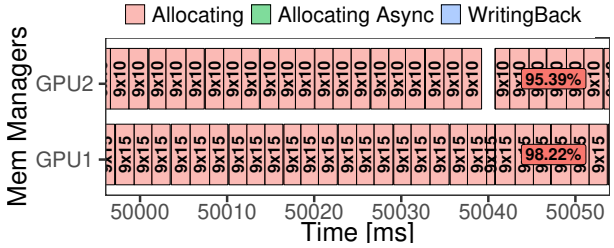


Fig. 8: Memory Manager states on a time frame of ≈50ms.

We employ the Block Residency plot to understand the previously described behavior. First, we select blocks in the initial iterations of the outermost loop of the tiled Cholesky

algorithm, with lower coordinates. They are expected to be more appropriate for being freed rapidly. Figure 9 shows that blocks 8×13, 9×13, 10×13, and 11×13 become present in all memory nodes at ≈50s. Shared copies of these blocks remain in each memory manager until the end of execution (at ≈140s). The presence of these blocks in all memory nodes indicates that StarPU is deciding not to free them.

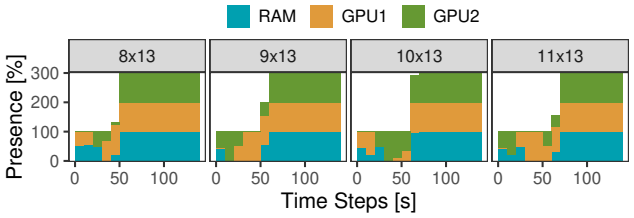


Fig. 9: Per-node block residency of [8-11]×13 coordinates.

All these insights gave us enough information to inspect decisions directly. We used `gdb` to check StarPU's functions that free unused memory blocks. StarPU all the time believed that it had free space on the GPUs. We also detected a huge difference when comparing the internal StarPU's used memory values to the ones given by the GPU resources monitor. This difference led us to the discovery that the CUDA function `cudaMalloc` could allocate more memory than the requested size and the runtime was not considering it. The function rounds the demanded memory to a device dependent page size, effectively causing internal memory fragmentation. Our GTX1080Ti has a fixed page size of 2 MB. Since our blocks are 960x960 (7200 KB each), every block allocation request in the GPU causes a loss of 992 KB (because it returns four pages to answer that request). That leads to loosing 1800 MB for the 60×60 tiles used in our experiments. StarPU was miss-calculating the GPU memory utilization and kept calling the expensive `cudaMalloc` function even without GPU memory. We then proposed a fix for StarPU and compared the Cholesky performance before and after our patch.

Figure 10 presents the performance comparison between two StarPU versions: original (red color) and corrected with our patch (blue). For both versions, we employ the same application and runtime configuration with the DMDA scheduler. We depict the application performance in GFlops on the Y-axis as a function of the input size, on the X-axis. The input size distribution has more points around and after the memory limit (marked by a vertical dashed line, calculated based on rounding behavior, number of blocks and the CUDA driver used memory) when the matrix size no longer entirely fits on the GPU memory. The dots represent the mean of ten executions and error bars are drawn based on a 99% confidence interval. We can see that the original version demonstrated a significant performance drop after the memory threshold, falling from the relatively constant ≈700 GFlops rate to poorer values. After our patch, the corrected version keeps the performance stable at the ≈700 GFlops rate. These results demonstrate the effectiveness of our fix, maintaining the program's scalability as the input size increases.

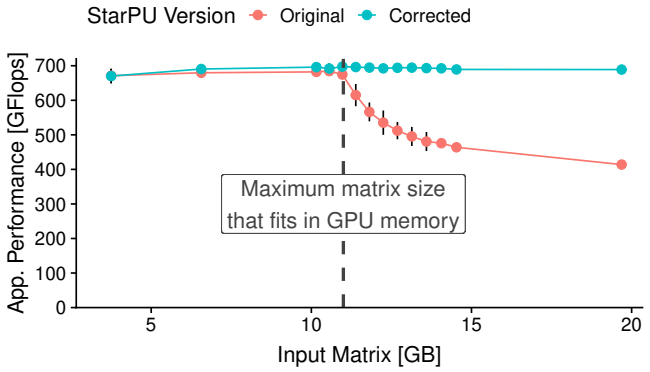


Fig. 10: Application performance (GFlops) as a function of the input size for two versions: the original and after our fix.

B. *Understanding the idle times on out-of-core use*

We conducted experiments using the out-of-core version of the dense Cholesky factorization of Chameleon, using the developer branch of StarPU. Our first experiment has the following configuration: block size of 960x960, with 20x20 tiles, and the `DMDAR` scheduler. Using StarPU environment variables, we artificially limited the `RAM` to 1.1 GB to stress the out-of-core support. Figure 11 depicts the behavior of five CPU and two CUDA workers. The initial phase of the execution presents idle times on all workers. We select a task preceded by a significant idle time to understand this poor behavior and why it is delayed. For such a purpose, we choose Task `1667` and highlight it with the red color (the right-most selected rectangle in the figure). The red line arriving at the chosen task indicates the two last dependencies to have been satisfied to execute Task `1667`. We can see that at least ≈500ms have passed between the execution of the last dependency and our selected task.

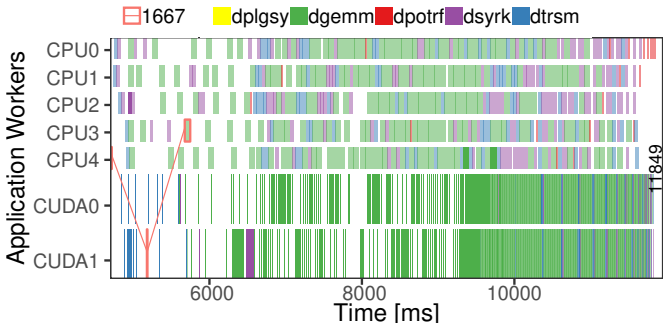


Fig. 11: Application workers using out-of-core, DMDAR scheduler, block size of 960x960 and 20x20 tiles.

Task `1667` uses the memory blocks 0×2, 0×9 and 2×9. We employ the detailed history of these memory blocks to identify if the reason behind the idle times in the initial phase of the execution is due to late data transfers. Figure 12 depicts

the behavior between 5000ms and 6000ms. Task 1667 is a dgemm operation marked as a green inner rectangle in all facets (one per memory block). This task starts exactly at 5676ms, while its last dependency finished at 5185ms: a vertical green line highlights this moment. The last task that executed on the same worker (CPU3, not shown), ended at time 5486ms and is marked as a vertical yellow line.

We can see in Figure 12 that when the last dependency ends, there are prefetch transfer requests created on blocks 0×9 and 2×9, and this request is satisfied on block 0×9 with a transfer. However, we can see that the Task 1667 only starts to be executed after the transfer of block 2×9, meaning that it was that memory block that was holding its execution. We also check that block 2×9 had a fetch transfer request at time 5486ms when the last job on the same worker ended. The time between this last job and the start of Task 1667 is considered idle time on the worker. Besides, the allocation request of block 2×9 on RAM was made almost 100ms after the transfer request. Reducing this gap could lead to less idle time in the worker. Finally, there are two allocation requests on RAM of block 2×9 because an allocation request can fail possibly due to the lack of fast (RAM, GPU) memory. In this case, StarPU proceeds to free memory blocks by moving a less critical block to the disk, for instance. StarPU developers are aware of these issues, and the next release might include a fix to reduce waiting time in an OOC experiment.

C. Performance impact of matrix generation order using OOC

When comparing the LWS and the DMDAR schedulers, under the same experimental conditions of the previous subsection, we noticed that the LWS scheduler was consistently faster than DMDAR, as it had a makespan of 10s while the DMDAR had 12s. These results were surprising since DMDAR is a much more sophisticated scheduler. It is supposed to reduce data transfers by running tasks close to the location of their input data.

We have employed the memory-aware animation snapshots (as detailed in Section IV-D) for each scheduler to understand the unexpectedly good performance of LWS. Figure 13 shows the memory snapshots when the first dpotrf task gets executed (on the 0×0 block coordinate – top left) for both LWS (bottom row) and DMDAR (top) schedulers. We can see that they are storing the initial input matrix differently. In the LWS, the input matrix is generated from top to bottom, so the required blocks are readily available in RAM. In the DMDAR, the matrix generation is inverted, forcing more data transfers from the disk to RAM at the beginning of the execution.

We use heatmaps, as shown in Figure 14, to demonstrate how in general the wrong matrix generation strategy may be harmful to overall performance. The heatmaps show that the upper side of the matrix on the LWS is generally much more present on RAM than the lower side. At the beginning of the execution, the critical low-coordinates blocks are already on RAM leading to fewer transfers from disk during the whole run. Therefore, the unnecessary data transfers of DMDAR justify its poor behavior when compared to the LWS scheduler.

D. CPU-only restricted RAM in a DMDAR×DMDAS comparison

We conduct CPU-only experiments, similar to the B and C cases, but with a very restrictive RAM size of 512MB. Our goal is to stress the data-awareness of the DMDAS and DMDAR schedulers. We identify that DMDAS is 60% [50s vs 30s] slower than DMDAR. One of the reasons is the different number of transfers done by the schedulers. While DMDAR did ≈1981 RAM-to-disk transfers, DMDAS did ≈3097, a increase of ≈50%.

To continue our understanding of the differences, Figure 15 shows ten representative block residence snapshots (horizontal facets) for both schedulers (top and bottom rows) out of two animations with many frames. We can see that there are differences on how block residence evolves for each of scheduler. DMDAR prioritizes executing tasks whose data are already available in memory (i.e., ready). Thus, by linear algebra dependencies, this scheduler tends to work on a linear/columnar way. Therefore, we can see the divisions of row and column on the visualizations. For example, we can see that the matrix is processed column-wise (the manually-added arrows highlight the behavior) in snapshots one to five. So on each column or row is preferable to be put on RAM.

On the other hand, DMDAS sorts tasks by priorities, considering the topological distance to the last task. This sorting forces a diagonal advance on the matrix, contradicting row/-column locality, a behavior that can be especially seen on snapshots three to eight (arrows highlight the behavior). This diagonal exploration suggests that although task priorities are useful to guide the scheduler towards the critical path, they should not necessarily be enforced too rigorously, to let the scheduler re-use ready data blocks for better locality.

VI. Known Limitations

We detail four known limitations of our methodology and implementation. (1) Our methodology is general but specifically tailored for the StarPU runtime and its applications. The views depend on traces enriched with events about memory operations only available, as far as we know, in the StarPU runtime. Other runtimes (such as for OmpSs [3] and OpenMP) must be instrumented to make them work with StarVZ. Our methodology is easier to extend for runtimes that divide the memory into blocks and use the MSI memory coherence protocol. (2) Although we use only one application (Chameleon), our views would work for any application that uses the StarPU runtime. For example, results of Section V-A benefit all StarPU applications with a similar memory block-based layout and problems larger than the GPU memory. The out-of-core case studies bring positive results that are exclusive to the Cholesky application. However, this understanding can lead to potential improvements to other applications that use the out-of-core capabilities of StarPU. (3) We focus on single node executions because even if we increase the number of nodes, the identified problems would still be there, just with more information to be processed. Problems surface in multi-GPU memory interactions, reinforcing the obstacles in memory management for heterogeneous platforms. (4) Our views require knowledge about both the runtime and the

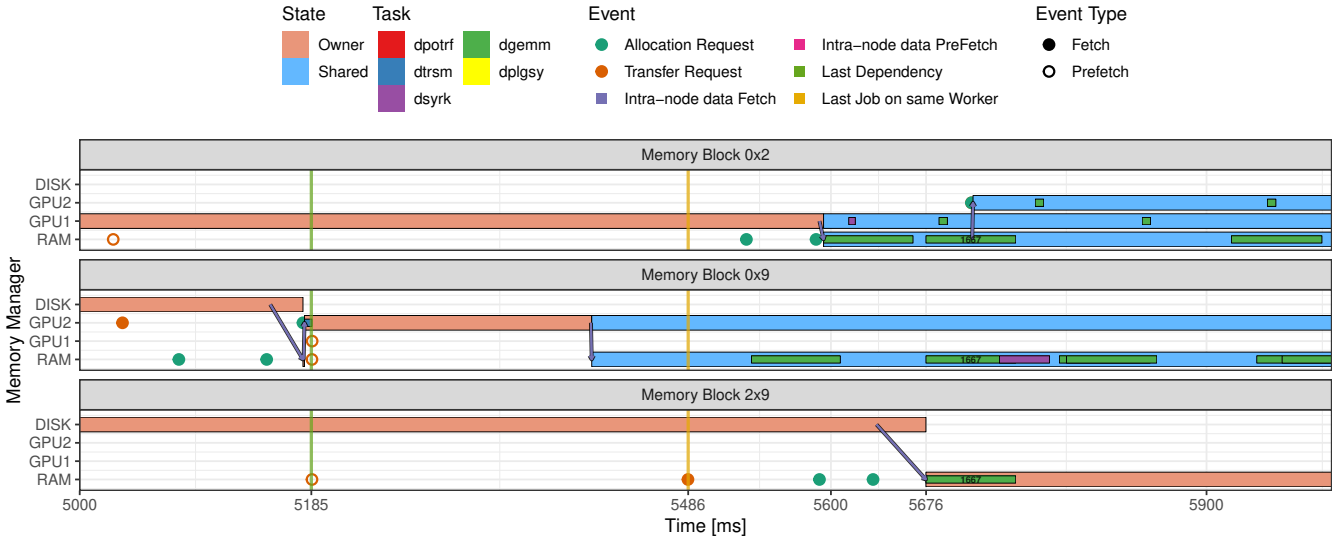


Fig. 12: Identifying delays in data transfer to satisfy the memory blocks necessary to run Task 1667.

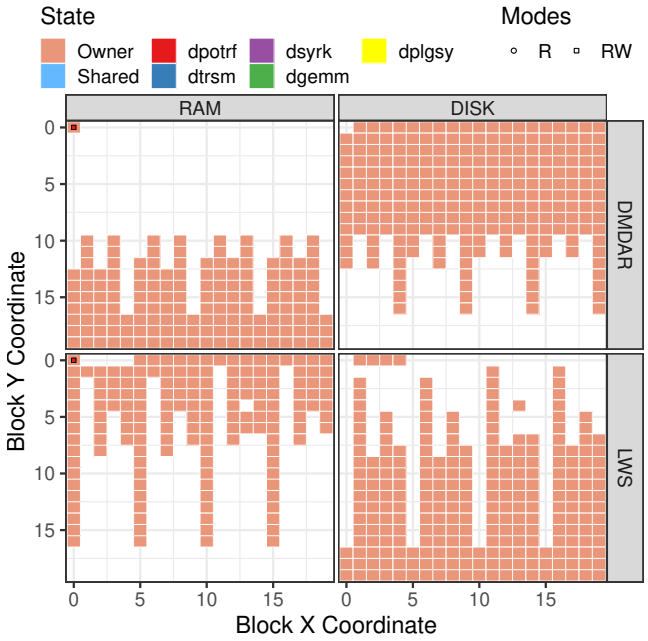


Fig. 13: Snapshots of block presence when application starts.

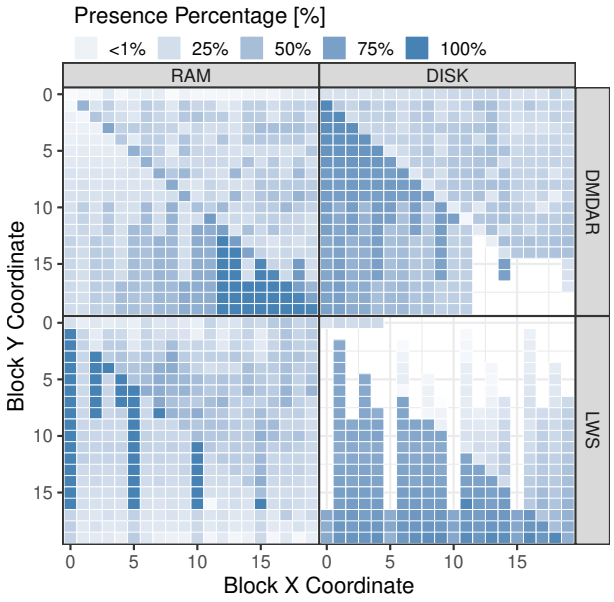


Fig. 14: Heatmaps showing block presence throughout the run.

application to make conclusions. However, we believe that our tool is intuitive and easy to use for researchers with some HPC experience. Our method in the hands of advanced analysts or HPC researchers could provide a broad understanding of the final application performance and how to improve it.

VII. Conclusion

We present a visual performance analysis methodology based on known visualization techniques (space/time, heatmaps, etc.) that are modified in a novel way to support the identification of data transfer inefficiencies both in the StarPU task-based runtime and application code. We investigate the performance of data transfer and out-of-core algorithms with a tile-based dense Cholesky factorization in four scenarios. (A) The wrong perception by StarPU of GPU memory utilization, issuing too many memory allocations requests that ultimately hurt performance. (B) Identifying significant idle times in applications with the out-of-core feature and providing rich details on the memory operations that caused it. (C) The identification of unexpected performance issues caused by input on StarPU schedulers under limited memory. (D) The explanation of the performance differences between two StarPU schedulers on a CPU-only execution with severe memory constraints.

The case studies indicate that our views are fundamental to identify the reasons behind performance problems related to

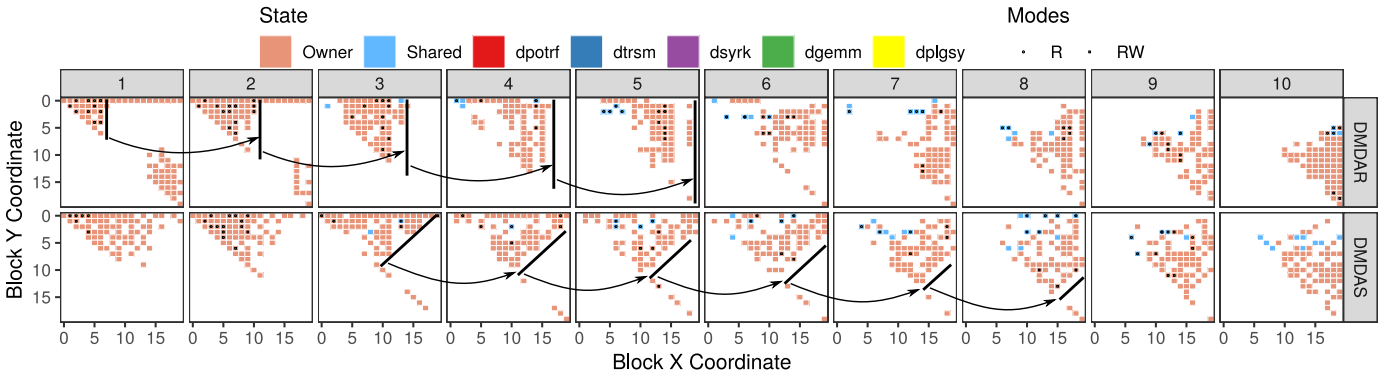


Fig. 15: Ten block residence snapshots (horizontal facets) for the DMDAR (top row) and the DMDAS (bottom row) schedulers.

memory operations. These panels lead us to understand how to fix memory operations of the StarPU runtime, and to know how data transfer policies affect scheduling algorithms. StarPU delivers gains of 66% (with multiple GPUs and CPUs) and better scalability for larger workloads.

Future work includes the analysis of data transfers on distributed StarPU executions, of other applications (such as sparse solvers, CFD codes), and OpenMP-based runtimes.

Acknowledgements

We would like to thank Arnaud Legrand for the insights and the discussions about this work. This study was financed by the "Coordenação de Aperfeiçoamento de Pessoal de Nível Superior" (CAPES) - Finance Code 001, the National Council for Scientific and Technological Development (CNPq), and the FAPERGS GreenCloud (16/488-9), the FAPERGS MultiGPU (16/354-8), the CNPq 447311/2014-0, the CAPES/Brafitec EcoSud 182/15, and the CAPES/Cofecub 899/18 projects. The companion material is hosted by CERN's Zenodo for which we are also grateful.

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of parallel and distributed computing*, vol. 37, 1996.

[2] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *IEEE Intl. Symposium on Parallel and Distributed Processing*, 2013.

[3] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: a proposal for programming heterogeneous multi-core architectures," *Paral. Proces. Letters*, vol. 21, 2011.

[4] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Conc. Comp.: Pract. Exp., SI:EuroPar'09*, vol. 23, 2011.

[5] E. Agullo, O. Beaumont, L. Eyraud-Dubois, J. Herrmann, S. Kumar, L. Marchal, and S. Thibault, "Bridging the gap between performance and bounds of cholesky factorization on heterogeneous platforms," in *IEEE Intl. Parallel and Distributed Processing Symp. Workshop*, 2015.

[6] L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J.-F. Méhaut, "Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures," *Concurrency and Computation: Practice and Experience*, p. 16, May 2015.

[7] L. Stanisic, E. Agullo, A. Buttari, A. Guermouche, A. Legrand, F. Lopez, and B. Videau, "Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers," in *The 21st IEEE International Conference on Parallel and Distributed Systems*, Melbourne, Australia, 2015.

[8] V. G. Pinto, L. M. Schnorr, L. Stanisic, A. Legrand, S. Thibault, and V. Danjean, "A visual performance analysis framework for task based parallel applications running on hybrid clusters," *Concurrency and Computation: Practice and Experience*, 2018.

[9] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-Aware Task Scheduling on Multi-Accelerator based Platforms," in *16th Intl. Conference on Parallel and Distributed Systems*, Shangai, 2010.

[10] S. Toledo, "A survey of out-of-core algorithms in numerical linear algebra," *Ext. Mem. Alg. and Vis.*, vol. 50, pp. 161–179, 1999.

[11] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs," in *GPU Computing Gems*, W. mei W. Hwu, Ed. Morgan Kaufmann, Sep. 2010, vol. 2.

[12] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems," *ACM Tr. Math. Softw.*, vol. 43, no. 2, 2016.

[13] M. Sergent, D. Goudin, S. Thibault, and O. Aumage, "Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System," in *21st Intl. Workshop on High-Level Paral. Prog. Models and Supportive Environments*, Chicago, USA, 2016.

[14] B. Haugen, S. Richmond, J. Kurzak, C. A. Steed, and J. Dongarra, "Visualizing execution traces with task dependencies," in *The 2nd Workshop on Visual Perf. Analysis*. New York, NY, USA: ACM, 2015.

[15] A. Huynh, D. Thain, M. Pericàs, and K. Taura, "DAGViz: A DAG Visualization Tool for Analyzing Task-parallel Program Traces," in *Proceedings of the 2nd Workshop on Visual Performance Analysis*, ser. VPA '15. New York, NY, USA: ACM, 2015, pp. 3:1–3:8.

[16] A. Muddukrishna, P. A. Jonsson, A. Podobas, and M. Brorsson, "Grain graphs: OpenMP performance analysis made easy," in *24th ACM SIGPLAN Symp. on Principles and Practice of Paral. Prog.*, 2016.

[17] R. Keller, S. Brinkmann, J. Gracia, and C. Niethammer, "Temanejo: Debugging of thread-based task-parallel programs in StarSs," in *Proc. of the 5th Intl. Workshop on Paral. Tools for High Performance Computing, September 2011, ZIH, Dresden*. Springer, 2012, pp. 131–137.

[18] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The Vampir performance analysis toolset," in *Proc. of the 2nd Intl. Workshop on Parallel Tools for High Performance Computing*. Springer, 2008, pp. 139–155.

[19] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A Tool to Visualize and Analyze Parallel Code," in *Proceedings of WoTUG-18: Transputer and occam Developments*, P. Nixon, Ed., 1995, pp. 17–31.

[20] K. Coulomb, M. Faverge, J. Jazeix, O. Lagrasse, J. Marcoueille, P. Noisette, A. Redondy, and C. Vuchener, "Visual trace explorer (ViTE)," Tech. Rep., 2009.

[21] K. E. Isaacs, P.-T. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, and B. Hamann, "Combing the communication hairball: Visualizing parallel execution traces using logical time," *IEEE Trans. on visualization and computer graphics*, vol. 20, no. 12, pp. 2349–2358, 2014.

[22] G. Ceballos, T. Grass, A. Hugo, and D. Black-Schaffer, "Analyzing performance variation of task schedulers with TaskInsight," *Parallel Computing*, vol. 75, pp. 11 – 27, 2018.

[23] M. Pericàs, A. Amer, K. Taura, and S. Matsuoka, "Analysis of data reuse in task-parallel runtimes," in *High Perf. Comp. Syst. Perf. Modeling, Bench. and Simul.* Springer Intl., 2014, pp. 73–87.